

1. [Introduction](#)
2. [Methodology and Design Choices](#)
3. [Basics of OCR](#)
4. [Tutorial](#)
5. [Results & Conclusion](#)
6. [Future Work](#)
7. [References](#)

Introduction

Optical character recognition, or OCR for short, is applicable to a variety of fields and problems. Take the example of recognizing the license plates of cars running red lights caught by speeding cameras. Using OCR, we might determine the characters on the license plate of the speeding car.

Initially, we were planning on doing a project along the lines of license plate recognition. By developing an application that could recognize the characters on a license plate, we would be able to automate the billing process for cars caught by speeding cameras, for example. However, as we searched the Internet for tutorials on how to perform optical character recognition (OCR), we had difficulty finding a fully comprehensive one that taught us how to train an algorithm on a custom dataset and then perform character recognition on similar images. We did find MNIST, a database of handwritten digits, but it does not include letters, nor does it pertain to the artificial text on license plates. As a result, we decided we would make our own tutorial.

Our main goal was to create a general step-by-step, easy-to-use tutorial to optical character recognition by performing basic techniques available in Python's OpenCV library on a simple data set, to be uploaded to Connexions as a contribution to the open education movement.

As a result, we decided that we would perform OCR on typed-text input in png and jpeg files as our valid input, with the letters of the alphabet (both capital and lowercase) and digits 0-9 as our classes of characters.

Methodology and Design Choices

We decided to create our tutorial in Python using the open source computer vision library, OpenCV, for its ease of use and minimal barrier to entry – Python is free, where Matlab is not, and Python is easier to learn than C++ for beginners. OpenCV, being open-source, is well documented in its functionality, though it lacks the kind of comprehensive tutorial we have developed.

When determining how to implement our tutorial, we decided we would keep the methods and training data simple. Our input consisted of clean images of typed text to teach the user how OCR can work for typed characters – the letters of the alphabet and numbers in this case. We decided that it would be simplest if we used images and characters with the correct orientation as well.

The feature vector we used was a vector of the average intensity values of each character after resizing it to a standard size because this was simple to implement in OpenCV.

For machine learning models, we went with the k-nearest neighbors (KNN) algorithm (with $k=1$) because KNN is easier to implement and explain than support vector machines. We settled on $k=1$, or finding the single nearest neighbor, because we found that it produced optimal results, probably because of the relatively small size of our training data set; we only trained on 15 fonts. Also, since we would ideally be training on a variety of fonts, the boundaries between clusters of characters of each class might not be quite as easily defined, especially in the case of similar characters like O and 0, which would lend to KNN being more effective than SVM.

Basics of OCR

For OCR, we need to assume an image has certain textual characteristics. For example, there is no point in using a picture of a tree as input to text recognition software.

After reading in an image, the first step to OCR involves preprocessing. Any picture the software reads in can be represented by a matrix of red-green-blue(RGB) values. Rather than deal with three variables, we can make this more computation friendly by converting from RGB to grayscale; instead of a matrix with three separate values in each cell, we now have a matrix of intensity values between 0 and 255.

Another issue is that the images dealt with in OCR are not necessarily properly oriented; they may be skewed, angled, or flipped. This incorrect orientation would make it more difficult for us to correctly classify characters. In order to remedy this, a typical preprocessing requires us to transform and translate the pixels within the image in an attempt to realign the text.

Depending on our inputs and assumptions, there are multiple options on what we will use as a filter. For our purposes, we will be looking at a few filters meant for edge detection, specifically Gaussian, Laplacian, and Sobel.

After filtering, we will be utilizing OpenCV's wide array of functions to detect our characters and identify them. The API provided will handle tasks such as defining the threshold of an edge and the actual edge detection. For classifying the characters, OpenCV has a machine learning algorithm, K-Nearest Neighbors that we capitalize on.

Tutorial

Preparation

All files referenced and other helpful information is posted on this [site](#) at the GitHub link. The GitHub link also has 3 Python files, which is our completed OCR software. This tutorial serves as a walkthrough of the code under recognitionScript.py, where more in-depth explanations are discussed in the provided .py files.

This tutorial uses Python 2.7, Numpy, and OpenCV for the software development portions. The instructions for downloading OpenCV for Linux can be found at [this link](#). For windows, go to to [opencv.org](#) For Numpy, visit [numpy.org](#) For Python, visit [python.org](#)

We assume the reader will have been exposed to Python before. If not, a general understanding of Python IO and the ability to look up Python documentation is preferable.

Preprocessing

For our tutorial, we will make a few assumptions for our input images. First off, the image will only have printed text. We also assume the image will be a png. For example, we will have pictures similar to the figure below.

```
1 2 3 4 5 6 7 8 9 0
Q W E R T Y U I O P A S D F G H J K L Z X C
V B N M
q w e r t y u i o p a s d f g h j k l z x c
v b n m
```

Because this is an introduction, there will not be information on transforming and translating, as this involves verbose algorithms beyond the scope of our knowledge.

Second, images may contain the full range of colors.

The first step is to read in the image we will be performing OCR on. OpenCV has the ability to do this with `imread()` function. We have made a wrapper function that does this and returns a copy, which will be necessary for some OpenCV functions

```
#Input:
#   path directory for a Image file
#Output:
#   Tuple containing the image object and a copy
def readImage(fileName):
    im = cv2.imread(fileName)
    imcopy = im.copy()
    print "Read file:", fileName
    return im, imcopy
```

Right now, `im` is an image, which can be represented as a matrix with pixels. Each pixel is some object with 3 values, red, blue, green(rgb). In order to make this compatible with future functions, we will convert the image to grayscale. This involves a function provided by OpenCV, which process the rgb values and replace the pixel with some intensity value. The next step is to convert the image into grayscale. Open Cv has a color converting function called `Cv2.cvtColor(image, code )` which can map one type of color to another based on the input code. For the purposes of converting our image into grayscale we use the parameter `Cv2.COLOR_BGR2GRAY` and call our the helper function `color2gray`, which returns the grayscaled image.

```
#Input:
#   Image object that is to be converted
#Output:
#   Converts the color image to grayscale
def color2gray(image):
    gray = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY) # convert color image to grayscale
    return gray
```

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

0 1 2 3 4 5 6 7 8 9

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

0 1 2 3 4 5 6 7 8 9

Above is our grayscale image.

Filtering

Next, we will need to apply a filter onto the image. Depending on the task, we could use many different filters to achieve a certain goal. We want to filter our image to increase our ability to detect edges; this allows us to identify characters in the image. Therefore, we will be covering the Gaussian, Laplacian, and Sobel filters.

Gaussian Filter

Using the following formula, we will create some n by n matrix to build the Gaussian filter.

$$g(x, y) = \frac{1}{2\pi\sigma^2} \cdot e^{-\frac{x^2+y^2}{2\sigma^2}}$$

The Gaussian filter is effective towards noisy signals, because of its characteristic as also being a low pass filter. By blurring the image to a small degree, we allow any sensitive edge detection algorithms to not mistake noise to be something of significance.

Laplace

In cases where we know that noise is not an issue, we can use the Laplacian filter. Rather than worry about reducing noise, the Laplace filter's purpose is to enhance edges by sharpening the image. The formula to describe the filter is

$$L(x, y) = \frac{\partial^2 I}{\partial x^2} + \frac{\partial^2 I}{\partial y^2}$$

Because this filter serves to enhance the features of an image, this also increases noise.

Sobel

Sobel filtering involves using two 3 by 3 matrices to convolve with the image in order to find the gradient of the image. Although this may be an inaccurate approximation, it proves effective for our needs.

$$\mathbf{G}_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathbf{A} \quad \text{and} \quad \mathbf{G}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathbf{A}$$

After testing with multiple files, we had come to the Gaussian filter was best suited for the current data sets. There is a convenient OpenCV function that does this, `Cv2.gaussianBlur( .. )`. With appropriate parameters: the width and height of the matrix that will be used to filter the image and the standard deviation of the Gaussian in the x and y directions (the greater standard deviation the less variance among the pixels after filter, i.e. greater blur)

```
#Input:
# Image object that is to be converted
#Output:
# Image objected blurred to the specified parameters
def gaussianBlur(image, width=5, height=5, xStdev=0, yStdev=0): # width and height has to be odd and positive
    gblur = cv2.GaussianBlur(image,(width,height),xStdev) # gaussian blur
    return gblur
```

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

0 1 2 3 4 5 6 7 8 9

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

0 1 2 3 4 5 6 7 8 9

Above is our blurred image.

Feature Detection

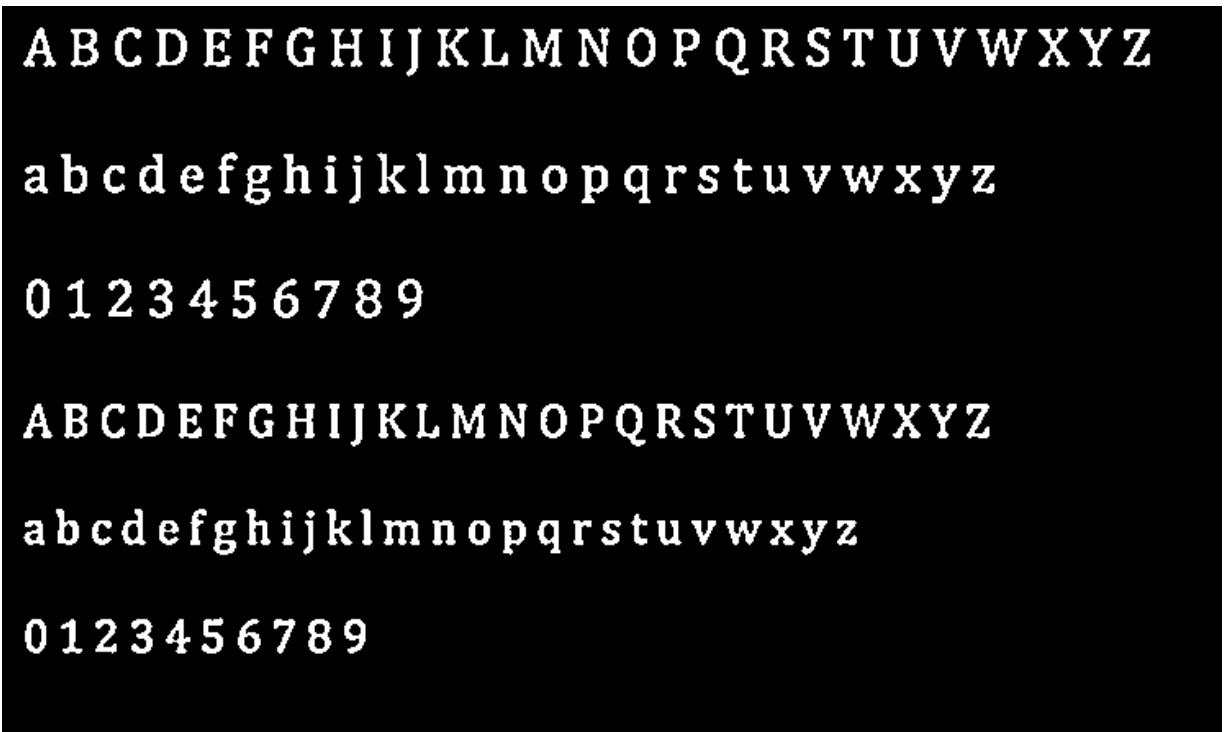
The next step is to determine where the edges in the image exist. After filtering, there needs to be some way to take each character and define its features in some measurable value. Under OpenCV, we used the `adaptiveThreshold` function to take our image and decide whether a certain intensity value in the image should be a 0 or 1. Effectively, our processed image is currently a matrix of binary values. We have the provided helper function.

```

# Input:
#   Image object that has been grayscale
# (Optional) Parameters:
#   MaxValue - max pixel value, 255 in this case
#   Adaptive Method - It decides how thresholding value is calculated.
#       cv2.ADAPTIVE_THRESH_MEAN_C : threshold value is the mean of neighbourhood area.
#       cv2.ADAPTIVE_THRESH_GAUSSIAN_C : threshold value is the weighted sum of neighbourhood values where weights are a gaussian window.
#   Threshold value- style of threshold, in this case Binary thresh
#       cv2.THRESH_BINARY
#       cv2.THRESH_BINARY_INV
#       cv2.THRESH_TRUNC
#       cv2.THRESH_TOZERO
#       cv2.THRESH_TOZERO_INV
#   Block Size - decides the size of neighbourhood area.
#   C - constant which is subtracted from the mean or weighted mean calculated.
#
# Output:
#   same type of image object but where each pixel has gone through binary
# Further Reading: http://docs.opencv.org/trunk/doc/py\_tutorials/py\_imgproc/py\_thresholding/py\_thresholding.html
def adapThreshold(image, MaxValue = 255, Method = cv2.ADAPTIVE_THRESH_GAUSSIAN_C, Threshold_Type = cv2.THRESH_BINARY_INV, Blocksize=11, c=2):
    thresh = cv2.adaptiveThreshold(image, MaxValue, Method, Threshold_Type, Blocksize, c)
    return thresh

```

In deciding whether a pixel meets the threshold there are two methods: we can use an adaptive mean filter or a Gaussian. We found a Gaussian was better. The threshold type **should** be `cv2.THRESH_BINARY_INV` which turns inverts the values-pixels deemed white become black and vice versa. This is because Open Cv's edge finding functions find white characters in black backgrounds.



Above shows the adaptive threshold of the image.

From here, we can use OpenCV's findContours method to find the edges. This will return coordinates, width, and height of a rectangle around a character. Given specific properties, some rectangles may be removed, resized, or combined to accomodate special cases.

```
# Input:
#   Image object that will be modified to find contours, will find white objects in this black file
#   (Optional) Parameters:
#   Approx Method
#       cv2.CHAIN_APPROX_SIMPLE: returns a simple amount of points i.e 4 for a box for the corners
#       cv2.CHAIN_APPROX_NONE: Returns all redundant points for contour i.e hundred for a box for points on edge
#   Retrieval Method
#       cv2.CV_RETR_EXTERNAL retrieves only the extreme outer contours. It sets hierarchy[i][2]=hierarchy[i][3]=-1 for all the contours.
#       cv2.CV_RETR_LIST retrieves all of the contours without establishing any hierarchical relationships.
#       cv2.CV_RETR_CCOMP retrieves all of the contours and organizes them into a two-level hierarchy. At the top level, there are external boundaries of
#       cv2.CV_RETR_TREE
# Output:
#   Contours: list of all the contours in the image
#   Hierarchy: output vector, containing information about the image topology and contours. It has as many elements as the number of contours.
# Further Reading:
#   http://docs.opencv.org/trunk/doc/py_tutorials/py_imgproc/py_contours/py_contours_begin/py_contours_begin.html?highlight=contours#what-are-contours
#   http://docs.opencv.org/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html?highlight=findcontours#cv2.findContours
def findContours(image, Contour_Retval_Md = cv2.RETR_LIST, Contour_Approx_Method = cv2.CHAIN_APPROX_SIMPLE):
    contours, hierarchy = cv2.findContours(image, Contour_Retval_Md, Contour_Approx_Method)
    return contours, hierarchy
```

Afterwards, we call the following two helper functions, findContoursAreas() and removeOverlaps(). findContoursAreas removes countours in the list contourlist that do not meet a specified minimum height. We use this to ignore small contours, like the tittles around ‘i’s and ‘j’s.

removeOverlaps goes through the rectangles list and returns a list of contours(rectangles) that do not overlap, returning the largest rectangle if overlap does occur. This is necessary because the list findContourAreas returns all countours in a image, even those that do define an actual character. For example contours that make up part of a letter like the “o” in p would be included in addition to the contour that we want that encloses the “p.” We then sort the list to our liking so we can read the remaining letter outlines left to right, top to bottom. We do this with trainingHelper.xsort(countour_list). It uses a simple algorithm to sort the rectangles into sorted rows and then into columns.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

0 1 2 3 4 5 6 7 8 9

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

0 1 2 3 4 5 6 7 8 9

Above shows the code discovering the region of interests.

Taking each rectangle, we will create a vector to hold the features each image has. First, we use a resize method to convert each rectangle into a n by n matrix of values. The values is determined by dividing the image to n by n cells and returning the average intensity of each cell. From here, we turn the $n \times n$ matrix into a row vector. We can then add other features to this vector in order to determine the character. For example, take the average of the top half and the bottom half of the image.

```

# Resize Region of Interest into a size*size matrix of intensity values
# The default is 10 by 10 matrix
roismall = cv2.resize(roi,(size,size))

# Resize Pixel Intensity Matrix into a vector
# default is a 100 value vector of intensities
roismall = roismall.reshape((1,size*size + num_Extra_Vals))

# Convert vector into a float
roismall = np.float32(roismall)

# Find nearest vector using the Model's nearest KNN algorithm
retval, results, neigh_resp, dists = model.find_nearest(roismall, k = 1)

# Character that the ROI was recognized as
string = str(chr((results[0][0])))

```

Classification

To classify an image, there needs to be a referenced database to compare the characters to. Our provided code solves this through the training method, which uses machine learning. Using multiple sets of pictures, we take an image, run it through all of the steps above and tell the program what the character should be. When finished, we have training data, which holds a histogram of vectors mapped to specific characters.

```

#Input: Sample file,Response File
#
#Output: KNearest neighbor Model made from learning from the two files
#
def train(samplesFile, responsesFile):
    samples = np.loadtxt(samplesFile,np.float32)
    responses = np.loadtxt(responsesFile,np.float32)
    responses = responses.reshape((responses.size,1))

    model = cv2.KNearest()
    model.train(samples,responses)
    return model

```

Back to the current image, we take its feature vector and use a k-nearest neighbor algorithm to determine which vector is most similar to the one being compared. From there, we see what the vector is mapped to and return what should be the actual letter or number.

a b c d e f g h i j k l m n o p q r s t u v w x y z

a b c d e f g h i j k l m n o p q r s t u v w x y z

0 1 2 3 4 5 6 7 8 9

a b c d e f g h i j k l m n o p q r s t u v w x y z

a b c d e f g h i j k l m n o p q t s t u v w x y z

0 1 2 3 4 5 6 7 8 9

And there you have it! Thank you for reading through the introduction of OCR. Feel free to look through the files, as there will be more resources to explain this topic.

Results & Conclusion

By the end of the project, we had developed an introductory tutorial to the processes and algorithms involved in OCR.

With the methods we implemented, we were able to achieve little to no error when testing on our training data, and when testing on fonts not included in our training data set but in one of the font families included in the training set, we were able to achieve about 85% accuracy.

<https://cnx.org/content/m52098/>

Future Work

In the future we hope to expand our tutorial to take into account the multitude of techniques that can be applied during each step of the OCR project, ranging from filtering and edge detection to feature extraction and classification.

More specifically, we would implement support vector machines and logistic regression as two different learning models in addition to k-nearest neighbors, and show users how the results compare using each of these models. Pertaining to the data we handle, we would like to incorporate handwritten characters into our training data and to introduce noise into our images of typed text (by printing them and then scanning the printed images). We would also like to create a confusion matrix for the multiple implementations of OCR we would like to develop to assess their performance.

And finally, we look forward to taking ELEC 345: Computer Vision to expand our knowledge of the topic.

References

[http://dasl.mem.drexel.edu/alumni/bGreen/www.pages.drexel.edu/_we
g22/edge.html](http://dasl.mem.drexel.edu/alumni/bGreen/www.pages.drexel.edu/_we
g22/edge.html)
[http://dasl.mem.drexel.edu/alumni/bGreen/www.pages.drexel.edu/_we
g22/can_tut.html](http://dasl.mem.drexel.edu/alumni/bGreen/www.pages.drexel.edu/_we
g22/can_tut.html)
<http://docs.opencv.org/>
[http://www.mathworks.com/help/vision/examples/automatically-
detect-and-recognize-text-in-natural-images.html](http://www.mathworks.com/help/vision/examples/automatically-
detect-and-recognize-text-in-natural-images.html)
<http://www.caam.rice.edu/~timwar/CAAM210/OCR.html>
<http://www.ele.uri.edu/~hansenj/projects/ele585/OCR/>